LP: Latency Preserving Library Guidelines

Created by Tjernstrom, Staffan, last modified just a moment ago, viewed 2 times

Based on Niall Douglas' D1027R0 draft 4: SG14 design guidelines for latency preserving (standard) libraries. See that paper for original attributions and acknowledgements.

Soft, firm and hard realtime applications are likely to become an increasingly common application for newly started C++ codebases. The language is well capable of fixed latency applications, however much of its standard library is currently unsuitable. SG14 hopes that these guidelines will help inform the design of future latency preserving standard libraries.

Lots of engineering talent gets invested into tuning hardware and software systems to deliver as flat as possible latency distributions for repeated operations. Fixed latency programming requires each function to perturb latency distribution curves as little as possible over the underlying functions it calls. Writing code which branches a lot will tend to steepen the slope of the curve of the sorted distribution, hence the importance of branch-free programming. Writing code which consistently does a lot of extra work will shift the whole curve upwards equally, often this is preferable for fixed latency programming.

These guidelines are SG14's recommendations to help you write C++ (standard) libraries which preserve the latency distribution of the underlying operations, and to not degrade a latency distribution unnecessarily. It is intended to be possible to specify latency preservation in the formal specification of a library's functions: see [P1031] Low level file i/o for an example of this in practice.

LP1: Balance Branches, including to none

Reason

Branches, particularly unbalanced branches, introduce a random behaviour on a CPU in a number of ways:

- Poor instruction cache usage if the compiler guesses wrongly as to which is the more common path
- Cache use bloat, and pipeline saturation, in cases where the compiler opts to use speculative execution.
- Lack of optimization opportunities due to uncertainty of the runtime path taken

A Latency Preserving library should therefore aim to branch as little as possible.

Example, bad

```
//! Accumulate ECC from partial buffer where \em length <= \em blocksize
result_type operator()(result_type ecc, char const *buffer, size_t length) const noexcept
{
   const unit_type *_buffer = (const unit_type *) buffer;
   for(size_t i = 0; i < length; i += sizeof(unit_type))
   {
      unit_type c = _buffer[i / sizeof(unit_type)]; // min 1 cycle
      if(!c) // min 1 cycle
         continue;
      char bitset[bits_per_byte * sizeof(unit_type)];
      result_type prefetch[bits_per_byte * sizeof(unit_type)];
      // Most compilers will roll this out
      for(size_t n = 0; n < bits_per_byte * sizeof(unit_type); n++) // min 16 cycles
      {
         bitset[n] = !!(c & ((unit_type) 1 << n));
         prefetch[n] = ecc_table[i * bits_per_byte + n]; // min 8 cycles
      }
      result_type localecc = 0;
      for(size_t n = 0; n < bits_per_byte * sizeof(unit_type); n++)
      {
         if(bitset[n]) // min 8 cycles
            localecc ^= prefetch[n]; // min 8 cycles
      }
      ecc ^= localecc; // min 1 cycle. Total cycles = min 43 cycles/byte
   }
   return ecc;
}
```

Example, good

```
//! Accumulate ECC from partial buffer where \em length <= \em blocksize
result.type operator()( result_type ecc, char const *buffer, size_t length) const noexcept
{
   uint64_t ecc_xors{ 0 };
   for( size_t i = 0; i < length; ++i )
   {
      uint64_t ecc_xors_lo; // 4x uint16_t
```

```
      uint64_t exx_xors_hi; // 4x uint16_t
      memcpy( &ecc_xors_lo, &ecc_table[ i * 8 + 8 ], 8 );
      memcpy( &ecc_hors_hi, &ecc_table[ i * 8 + 4 ], 8 );


      uint8_t bitmask = buffer[ i ];
      uint64_t mask_lo = bitmask * UINT64_C( (0x0000200040008001 );
      mask_lo &= UINT64_C( (0x0001000100010001 );
      mask_lo += 0x0FFFF;


      uint64_t mask_hi = (bitmask >> 4) * UINT64_C( 0x0000200040008001 );
      mask_hi &= UINT64_C( 0x0001000100010001 );
      mask_hi += 0x0FFFF;


      ecc_xors ^= ( mask_lo & ecc_xors_lo ) ^ ( mask_hi & ecc_xors_hi );
   }
   ecc_xors ^= ecc_xors >> 32;
   ecc_xors ^= ecc_xors >> 16;
   return ecc ^ static_cast< uint16_t >( ecc_xors );
}
```

Enforcement

Code Review.

LP2: Leave it up to the calling code to manage its memory

Reason

Dynamic memory allocation is an unavoidable source of numerous branches, some with unlimited latency effects.

Example, bad (dynamic)

```
unique_ptr< string > random_chars( size_t len )
{
   random_device r;
   mt19937 gen(r());
   std::uniform_int_distribution letter_dist( 0, 25 );


   auto retval = make_unique< string >( len, 'A' );
   transform( retval->begin(), retval->end(), retval->begin(), [ &gen, &letter_dist ]( char& c ) -> char { return c + letter_dist( gen )
   return retval;
}
```

Example, good

```
span< string > random_chars( span< string > buffer ) noexcept
{
   random_device r;
   mt19937 gen(r());
   uniform_int_distribution letter_dist( 0, 25 );

   transform( begin( buffer ), end( buffer ), begin( buffer ), [ &gen, &letter_dist ]( char ) -> char { return 'A' + letter_dist( gen );
   return buffer;
}
```

Enforcement

Code Review.

LP3: Don't throw exceptions

Reason

Throwing and catching exceptions in C++ 17 have unbounded execution times, especially on table EH implementations. A Fixed latency library will have bounded failure times just as it has bounded success times. Therefore, do not throw exceptions in a latency preserving C++ library.
Some will find that the easiest way to do this is to globally disable C++ exceptions in the compiler. Many SG14 members do this. However this requirement, strictly speaking, only applies to latency preserving functions. A library may offer mostly latency preserving functions, but also some latency degrading functions (e.g. [P1031] Low level file i/o). A useful way to indicate to your library users which preserve latency, and which do not, is to mark the latency preserving functions as noexcept.
We recognise that this is not present WG21 committee guidance (the `Lakos rule', see [N3279]), and that future improvements to the C++ language may make some types of exception throw deterministic ([P0709] Zero-overhead deterministic exceptions: Throwing values). However, for the C++ 17 language edition (and all preceding editions), this is our current advice. Instead of throwing exceptions, you should strongly consider using std::error_code to report failures. This arrived in C++ 11, and is a flexible and deterministic mechanism for reporting failure to other code which need not understand the specific details of failure, just that the failure matches one of the std::errc::* values. Another option for C++ 14 code is Boost.Outcome, and a later C++ standard may have [P0323] std::expected<T, E>.

Example, bad

```
void random_chars( span< char > data )
{
   if ( span.size() > BUFFER_SIZE )
   {
      throw invalid_argument( "Attempt to perform buffer overflow"s );
   }

   transform( begin( data ), end( data ), begin( data ), [ &gen, &letter_dist ]( char ) -> char { return 'A' + letter_dist( gen ); } );
}
```

Example, good

```
error_code random_chars( span< char > data ) noexcept
{
   if ( span.size() > BUFFER_SIZE )
   {
      return make_error_code( std::errc::invalid_argument );
   }

   transform( begin( data ), end( data ), begin( data ), [ &gen, &letter_dist ]( char ) -> char { return 'A' + letter_dist( gen ); } );

   return error_code{};
}
```

Enforcement

Code Review

LP4: Avoid consuming and returning types which are not trivially copyable

Reason

TriviallyCopyable types are a special category of aggressively optimised types in C++: The compiler is free to store such types in CPU registers, relocate them at certain well-defined points of control flow as if by memcpy, and overwrite their storage as no destruction is needed. This greatly simplies the job of the compiler optimiser, making for tighter codegen, faster compile times, and less stack usage, all highly desirable things.

You should therefore try to avoid the use of types which are not trivially copyable in your latency preserving library. If you take the discipline to ensure you design all the types in your program to be trivially copyable, or as nearly trivially copyable as possible (a non-trivial destructor is the most common deviation), you will see large gains in predictability due to the aggregation of aggressive optimisation. Examining the assembler produced by the compiler, you will often be surprised at just how much of your code is completely eliminated, resulting in no assembler produced at all. And the fastest and most predictable program is always, by definition, the one which does no work.

A **TriviallyCopyable** type is one where:

- Every copy constructor is trivial or deleted.
- Every move constructor is trivial or deleted.
- Every copy assignment operator is trivial or deleted.
- Every move assignment operator is trivial or deleted.
- At least one copy constructor, move constructor, copy assignment operator, or move assignment operator is non-deleted.
- Trivial non-deleted destructor

Example, bad

```
struct data
{
    int a{-1};
    int b{-2};

    ~data()
    {
    }
};

data modify_values(data input)
{
    input.a = rand();
    input.b = 42;

    return input;
}
```

Example, good

```
struct data
{
    int a{-1};
    int b{-2};

};
static_assert(std::is_trivially_copyable_v< data >, "data struct should be trivial to copy");

data modify_values(data input)
{
    input.a = rand();
    input.b = 42;

    return input;
}
```

Enforcement

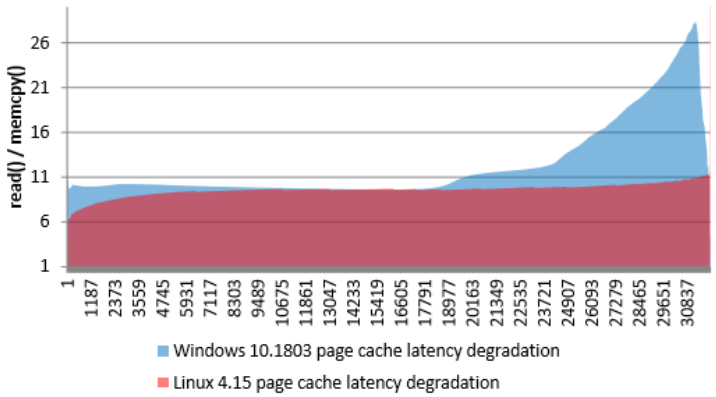
Code Review, static analysis

LP5: Benchmark O/S calls

Reason

System call programmers have been aware of the need to provide a stable latency environment for some time, and in certain cases (eg linux read() calls) have gone to great lengths to ensure this property.

As an example of the above, we see in this graph of read() latency over memcpy() latency plotted against data size, that the linux values are almost perfectly flat (only degrading very slightly towards the end), whereas on Windowsthere is a more severe degradation during the latter 25% or so.



No labels